Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №3

По дисциплине: «Языковые процессы интеллектуальных систем»

# Тема: «Проектирование лексического анализатора с использованием регулярных выражений»

Выполнил:

Студент 4 курса

Группы ИИ-21

Кирилович А. А.

Проверил:

Монтик Н.С.

Брест 2024

**Цель работы:** изучение основных понятий теории регулярных грамматик, ознакомление с назначением и принципами работы лексических анализаторов (сканеров), получение практических навыков построения сканера на примере заданного простейшего входного языка.

Вариант 17. Входной язык содержит многоуровневые арифметические выражения, разделенные символом; (точка с запятой). Выражения могут включать идентификаторы, числовые константы, скобки для задания приоритетов, а также функции (например, sin, cos). Операции могут включать сложение, вычитание, умножение и деление.

import 'dart:io';

class TokenType {

  static const String IDENTIFIER = 'IDENTIFIER';    // Идентификаторы

  static const String NUMBER = 'NUMBER';            // Числовые константы

  static const String OPERATOR = 'OPERATOR';        // Операторы (+, -, \*, /)

  static const String LPAREN = 'LPAREN';            // Левая скобка (

  static const String RPAREN = 'RPAREN';            // Правая скобка )

  static const String SEMICOLON = 'SEMICOLON';      // Точка с запятой ;

  static const String FUNCTION = 'FUNCTION';        // Функции (sin, cos)

  static const String COMMENT = 'COMMENT';          // Комментарии

  static const String CONSTANT = 'CONSTANT';        // Константы (e, pi)

}

class Token {

  String type;

  String value;

  Token(this.type, this.value);

  @override

  String toString() {

    return 'Token(type: $type, value: "$value")';

  }

}

class Lexer {

  String input;

  List<Token> tokens = [];

  Lexer(this.input);

  final RegExp \_numberPattern = RegExp(r'^\d+(\.\d+)?');

  final RegExp \_identifierPattern = RegExp(r'^[a-zA-Z\_][a-zA-Z\_0-9]\*');

  final RegExp \_operatorPattern = RegExp(r'^[\+\-\\*/=]');

  final RegExp \_whitespacePattern = RegExp(r'^\s+');

  final RegExp \_commentPattern = RegExp(r'^//.\*');

  final RegExp \_parenthesisPattern = RegExp(r'^[()]');

  final RegExp \_semicolonPattern = RegExp(r'^;');

  bool isAtEnd(int position) => position >= input.length;

  List<Token> tokenize() {

    int position = 0;

    while (!isAtEnd(position)) {

      String remainingInput = input.substring(position);

      final whitespaceMatch = \_whitespacePattern.firstMatch(remainingInput);

      if (whitespaceMatch != null) {

        position += whitespaceMatch.end;

        continue;

      }

      final numberMatch = \_numberPattern.firstMatch(remainingInput);

      if (numberMatch != null) {

        tokens.add(Token(TokenType.NUMBER, numberMatch.group(0)!));

        position += numberMatch.end;

        continue;

      }

      final commentMatch = \_commentPattern.firstMatch(remainingInput);

      if (commentMatch != null) {

        position += commentMatch.end;

        continue;

      }

      final identifierMatch = \_identifierPattern.firstMatch(remainingInput);

      if (identifierMatch != null) {

        String value = identifierMatch.group(0)!;

        if (value == 'sin' || value == 'cos') {

          tokens.add(Token(TokenType.FUNCTION, value));

        } else if (value == 'e' || value == 'pi') {

          tokens.add(Token(TokenType.CONSTANT, value));

        } else {

          if (value.length > 32) {

            throw FormatException("Ошибка: Идентификатор '$value' превышает 32 символа.");

          }

          tokens.add(Token(TokenType.IDENTIFIER, value));

        }

        position += identifierMatch.end;

        continue;

      }

      final operatorMatch = \_operatorPattern.firstMatch(remainingInput);

      if (operatorMatch != null) {

        tokens.add(Token(TokenType.OPERATOR, operatorMatch.group(0)!));

        position += operatorMatch.end;

        continue;

      }

      final parenthesisMatch = \_parenthesisPattern.firstMatch(remainingInput);

      if (parenthesisMatch != null) {

        String paren = parenthesisMatch.group(0)!;

        if (paren == '(') {

          tokens.add(Token(TokenType.LPAREN, paren));

        } else if (paren == ')') {

          tokens.add(Token(TokenType.RPAREN, paren));

        }

        position += parenthesisMatch.end;

        continue;

      }

      final semicolonMatch = \_semicolonPattern.firstMatch(remainingInput);

      if (semicolonMatch != null) {

        tokens.add(Token(TokenType.SEMICOLON, semicolonMatch.group(0)!));

        position += semicolonMatch.end;

        continue;

      }

      throw FormatException('Неожиданный символ: ${remainingInput[0]} на позиции $position');

    }

    return tokens;

  }

}

void analyzeFile(String filename) {

  File file = File(filename);

  String inputText = file.readAsStringSync();

  Lexer lexer = Lexer(inputText);

  List<Token> tokens = lexer.tokenize();

  print('${'Тип токена'.padRight(20)}${'Значение'.padRight(40)}');

  print('-' \* 60);

  for (var token in tokens) {

    print('${token.type.padRight(20)}${token.value.padRight(40)}');

  }

}

void main() {

  String filename = 'input.txt';

  try {

    analyzeFile(filename);

  } catch (e) {

    print('Лексическая ошибка: $e');

  }

}

**Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание**

**Вывод:** в ходе выполнения лабораторной работы изучил основные понятия теории регулярных грамматик, ознакомился с назначением и принципами работы лексических анализаторов (сканеров), получил практические навыки построения сканера на примере заданного простейшего входного языка.